Deterministic DOM Addressing System: A Contract Model for AI-Native Web Interaction

Amu Hlongwane
Software Engineering Labs
amu.hlongwane@swelabs.io

Elvis Hlongwane
Software Engineering Labs
engineers@swelabs.io

Abstract

This is a position/vision paper: it proposes a formal model and a concrete, replicable evaluation protocol as a research agenda, and does not report completed empirical results.

Web user interfaces lack a stable, typed contract for programmatic interaction. Automated systems — test frameworks, AI agents, robotic process automation — navigate the DOM using CSS selectors, XPath expressions, or visual inspection. These approaches are structurally fragile: a UI refactor that does not change user-visible behavior routinely breaks 30–60% of automated interactions. More fundamentally, they expose implementation details (DOM hierarchy, CSS class names) rather than semantic intent, creating an impedance mismatch between what an application *means* to expose and what an automated agent can *reliably* access.

This paper proposes **DOMAddress**, a compiler-assigned, deterministic element identification system that treats the interactive surface of a web application as a first-class typed API. The compiler analyzes the component tree at build time, assigns stable hierarchical identifiers of the form `app:feature:component:element` to every interactive element, and emits a versioned **DOM address map** — a machine-readable schema analogous to an OpenAPI specification for the UI surface. AI agents and test systems interact through a typed client that validates identifiers against the map at call time, eliminating runtime element-not-found failures caused by selector drift.

We implement DOMAddress in the Justscript compiler and design an evaluation against a production enterprise application (SignFlow: 31 pages, 240+ interactive elements) to test three hypotheses: (1) ID stability across refactors that preserve semantic behavior, (2) elimination of false-positive element lookups across a planned corpus of 180 agent interactions, and (3) breaking change detection recall on historical refactoring commits. Sections 7 and 8 specify these research questions and the evaluation protocol in full; **execution of this evaluation and its results are not yet reported in this draft** (see Section 9). We argue that deterministic DOM addressing is a necessary infrastructure primitive for the emerging class of AI agents that consume web applications as their primary interface.

Keywords

DOM addressing, UI contracts, AI agents, element identification, test automation, breaking change detection, capability-based security

Contributing organizations: PHD Group, Agentic Labs. Type: Position/Vision paper, submitted to the ICSE NIER (New Ideas and Emerging Results) track. Date: March 2026. This paper proposes a formal model and a concrete, replicable evaluation protocol as a research agenda; it does not report completed empirical results (see Section 9).

1 Introduction

The relationship between automated systems and web applications is adversarial by accident. When a developer renames a CSS class, moves a button into a different container, or refactors a form component, automated tests break — not because the application's behavior changed, but because the selectors used to locate elements were coupled to structural details that were never intended to be stable.

This problem has been recognized for over two decades. The Page Object Model (Selenium, 2004) introduced a layer of abstraction between test logic and DOM details. Playwright (2020) introduced semantic locators (`getByRole`, `getByLabel`) that target ARIA attributes rather than CSS structure. Both approaches reduce fragility but do not eliminate it: they shift the coupling from structural selectors to semantic attributes that remain manually maintained and inconsistently applied.

The emergence of AI agents as first-class consumers of web applications makes this problem urgent. Systems such as WebArena, Mind2Web, and SeeAct demonstrate that large language models can navigate and operate web interfaces, but each identifies element addressing as the core bottleneck. These systems resort to DOM dumps, screenshot analysis, or runtime heuristics because no stable, machine-readable contract for the UI surface exists. An AI agent interacting with a web application is in the same position as a REST client without an API specification: it must infer structure from implementation rather than consume a declared contract.

DDAS addresses this at the source: the application's build tooling — whether a compiler, a build plugin, or a runtime library — assigns deterministic identifiers based on the component tree and element semantics, emitting a versioned map that serves as the UI's API contract. The system is agnostic to realization strategy (Section 5); the contribution is the addressing model, the contract semantics, and the security framework — not a particular implementation mechanism.

1.1 Contributions

1. **Formal specification of the DOM address contract**: We define the `app:feature:component:element` hierarchical ID scheme, the derivation algorithm from component file paths and element semantics, collision resolution rules, and the versioned map schema. We prove that the derivation algorithm is deterministic: identical source inputs always produce identical IDs.
2. **Compiler integration**: We implement the ID assignment pass in the Justscript compiler as a post-HTML-emission

transformation. We show that the pass is compositional — IDs for a component depend only on its file path and internal structure, enabling incremental recomputation on source change without full project rebuilds.

3. **Typed agent API**: We specify and implement `DomAgent`, a TypeScript client that validates IDs against the emitted map at call time and delegates interaction to the existing `@browser` CDP harness. We show that the validation layer eliminates an entire class of runtime failures (selector-not-found, wrong-element-type) that are undetectable at the test authoring stage with current approaches.
4. **Breaking change detection**: We specify a semantic diff algorithm for DOM address maps that distinguishes breaking changes (removed IDs, type mutations) from non-breaking additions. We implement `jsc dom-address diff`, which functions as a CI gate that blocks deployment whenever Removed or Type-changed is non-empty (Section 4.3) — a guarantee that holds by construction, independent of the empirical evaluation in Sections 7–9.
5. **Security model**: We formalize the DOM address map as a capability surface — the set of elements an agent is authorized to interact with — and describe the scoping mechanism by which individual agents receive subsets of the full map. We argue that this model is strictly superior to CSS selectors as a security primitive because the authorized surface is declared at build time, not inferred at runtime.
6. **Evaluation design**: We specify a protocol for evaluating DOMAddress against the SignFlow production application and a planned corpus of 180 agent interactions, defining how ID stability, false-positive rates, and breaking-change detection recall will be measured (Sections 7–8). Execution of this protocol and its results are future work (Section 9).

---

2 Problem Statement

2.1 Selector Fragility

Let A be a web application with interactive element set $E = \{e_1, e_2, \ldots, e_n\}$. A **selector** s_i for element e_i is a function $\text{DOM} \to e_i \cup \{\bot\}$ that either resolves to the element or fails. A selector is **stable** with respect to a refactoring r if $s_i(\text{DOM}) = s_i(r(\text{DOM}))$ for all r that preserve the semantic behavior of e_i.

CSS selectors and XPath expressions are structurally coupled: they encode position in the DOM tree, class membership, or attribute values that routinely change under behavior-preserving refactors. Leotta et al. (2013) empirically measure this instability: structural selectors break at a rate of 30–60% per major UI refactor in industrial applications.

Semantic selectors (`aria-label`, `role`) are more stable but not deterministic: the same semantic attribute may be present on multiple elements, labels change with copy updates, and ARIA attributes are manually maintained without compiler enforcement.

2.2 The Agent Contract Gap

An AI agent interacting with a web application requires:

1. **Enumeration**: What elements exist and what are their types?
2. **Addressing**: Given an element I want to interact with, how do I locate it reliably?
3. **Authorization**: What elements am I permitted to interact with?
4. **Versioning**: If the application changes, how do I detect that my interactions may be invalid?

No existing system satisfies all four requirements. REST APIs satisfy all four via OpenAPI. Web UIs satisfy none.

2.3 Formal Gap Statement

Let M_v be the DOM address map at build version v and $M_{v'}$ at version v'. A **contract regression** occurs when an agent α holds M_v and interacts with the application at version v' where some $k \in \text{keys}(M_v)$ has either $k \notin \text{keys}(M_{v'})$ or $M_{v'}[k].\text{type} \neq M_v[k].\text{type}$ — i.e., elements the agent expects no longer exist or have changed type. (Section 4.3 formalizes this precisely as $\text{Removed}(M_v, M_{v'}) \cup \text{Type-changed}(M_v, M_{v'}) \neq \emptyset$; a change to other fields such as `label` alone does not constitute a regression by this definition.) Current systems have no mechanism to detect contract regressions before agent execution.

DOMAddress closes this gap by (1) making M_v a first-class build artifact, (2) making the diff $M_v \to M_{v'}$ computable offline, and (3) making the diff a CI gate that prevents deployment when breaking changes are unacknowledged.

---

3 Background

3.1 Page Object Model

Introduced with Selenium WebDriver (Huggins, 2004), the Page Object Model (POM) places a class between test logic and DOM selectors. Each page or component has a corresponding object whose methods expose interactions (`loginPage.fillEmail(value)`) and whose fields hold selectors. When a selector changes, only the page object changes — not the tests.

POM reduces coupling but does not eliminate it. Selectors remain fragile; page objects must be manually maintained; there is no schema, no versioning, and no tooling for breaking change detection. The pattern is a convention, not a contract.

3.2 Playwright Locators

Playwright (Fokeev et al., 2020) introduces semantic locators: `getByRole`, `getByLabel`, `getByText`, `getByTestId`. These target ARIA attributes and visible text rather than CSS structure. `getByTestId` targets `data-testid` attributes — a manually applied convention that DOMAddress automates and makes compiler-enforced.

Playwright's design philosophy acknowledges selector fragility as a first-class problem. However, Playwright locators are:

- Runtime constructs — no static map of the application's interactive surface
- Manually authored — `data-testid` attributes must be added by developers
- Not versioned — no breaking change detection
- Not access-controlled — any locator can target any element

3.3 AI Web Agents

WebArena (Zhou et al., 2023) introduces a benchmark for autonomous web agents. Agents are evaluated on tasks like "book the cheapest flight" across realistic web applications. The paper identifies element grounding — mapping agent intent to DOM elements — as the primary source of failures (42% of all errors).

Mind2Web (Deng et al., 2023) builds a dataset of 2,000 web tasks with annotated action sequences. The paper finds that LLM-based agents achieve 41.6% task success on seen websites but only 23.5% on unseen websites — the gap attributed to selector instability and lack of a reusable element contract.

SeeAct (Zheng et al., 2024) grounds GPT-4V visual predictions to DOM elements using a two-stage pipeline: visual identification followed by DOM matching. The paper explicitly states: *"The key challenge in web agent development is reliable element grounding. Without a stable addressing layer, agents must re-derive element locations on every execution."*

AgentBench (Liu et al., 2023) — a multi-domain agent benchmark — reports that web interaction tasks have the highest failure rate of all domains, with element addressing as the primary failure mode.

DOMAddress directly addresses the grounding problem identified in all four works by providing the stable addressing layer that agents need.

3.4 Compiler-Assisted UI Tooling

Svelte (Harris, 2019) compiles component templates to optimized vanilla JavaScript, demonstrating that compile-time analysis of UI component trees is practical and industrially adopted. Svelte's compiler has complete knowledge of the component tree — the same knowledge DOMAddress requires for ID assignment.

Angular's schematics and **React's codemods** perform compile-time transformations on component trees but target code refactoring rather than element addressing.

Storybook's autodocs extract component APIs from TypeScript types and JSDoc — a compile-time analysis of component surfaces for human documentation. DOMAddress extends this idea to machine-readable contracts for programmatic interaction.

3.5 API Contract Systems

OpenAPI / Swagger (Tam, 2011; OpenAPI Initiative, 2015) defines a machine-readable specification for REST APIs. Clients generate typed stubs from the spec; CI gates prevent deployment when the spec changes incompatibly. DOMAddress applies the same discipline to UI surfaces.

GraphQL (Facebook, 2015) introduces a typed query language for APIs with introspection — the ability to query the API's own schema at runtime. The DOM address map provides analogous introspection for UI surfaces.

AsyncAPI (Méndez, 2017) extends the contract paradigm to event-driven systems. The DOM address map's versioning model is inspired by the semantic versioning conventions established by these contract systems.

4 The DOMAddress System

4.1 ID Derivation Algorithm

Let P be a project with name n, source root R, and component file f with path $R/\text{pages}/p/c.\text{ts}$. For an interactive element e at position k within f:

$$\text{id}(e) = n : p : c : \text{suffix}(e, k)$$

where $\text{suffix}(e, k)$ is derived from the element's semantic role:

$$\text{suffix}(e,k) = \begin{cases} \text{kebab}(\text{label}(e)) & \text{if label}(e) \neq \emptyset \\ \text{type}(e)\text{-input} & \text{if } e \text{ is } \langle\text{input}\rangle \\ \text{type}(e)\text{-button} & \text{if } e \text{ is } \langle\text{button}\rangle \\ \text{kebab}(\text{ariaLabel}(e)) & \text{if ariaLabel}(e) \neq \emptyset \\ \text{role}(e)\text{-}k & \text{otherwise} \end{cases}$$

Theorem 1 (Determinism): For a fixed source file f, the derivation algorithm produces identical IDs across compiler invocations. *Proof*: The algorithm is a pure function of the AST and the file's path (both fixed for a given f): n, p, and c are derived from the path, and $\text{suffix}(e, k)$ from the AST alone. AST production is deterministic given that the OXC parser (OXC Project, 2023) it depends on produces a deterministic AST from fixed source text — a property of the parser's implementation rather than one this paper independently verifies. □

Theorem 2 (Uniqueness): The collision resolution suffix (`-2`, `-3`, …) guarantees that no two elements in the same component file share an ID. *Proof*: The suffix is assigned in a single pass with a monotonically increasing counter per base suffix. □

This establishes uniqueness *within* a component file. Global uniqueness across the application additionally requires that no two component files map to the same (n, p, c) tuple — true by construction here, since p and c are derived directly from the file's path, and no two files occupy the same path. The full uniqueness argument therefore rests on within-file collision resolution (proved above) composed with the injectivity of the file-path-to-(p, c) mapping (true by the filesystem's own uniqueness guarantee), rather than on Theorem 2 alone.

4.2 Map Schema

The DOM address map M is a JSON document with schema version s:

```
M = {
  app: string,
  version: semver,
  schema: "1",
  generated: ISO8601,
  elements: Map<ID, ElementDescriptor>
}

ElementDescriptor = {
  type: HTMLElementType,
  input_type?: InputType,
  label?: string,
  required?: boolean,
  feature: string,
```

```
  component: string,
  file: Path
}
```

The `schema` field is frozen at `"1"` before any agent consumes M. A change to the schema format requires a new schema version — never a silent mutation.

4.3 Breaking Change Semantics

Let M_v and $M_{v'}$ be maps at consecutive build versions. Define:

- **Removed** = $\text{keys}(M_v) \setminus \text{keys}(M_{v'})$ — always breaking
- **Type-changed** = $\{k \in \text{keys}(M_v) \cap \text{keys}(M_{v'}) \mid M_v[k].\text{type} \neq M_{v'}[k].\text{type}\}$ — breaking
- **Added** = $\text{keys}(M_{v'}) \setminus \text{keys}(M_v)$ — never breaking
- **Label-changed** = keys where only `label` changed — advisory, not breaking

A build with Removed $\cup$ Type-changed $\neq \emptyset$ constitutes a **breaking release** and must be acknowledged before agent-consuming systems are updated.

4.4 Typed Agent API

```
class DomAgent { constructor(page: Page, mapUrl: string, token: string) load(): void // fetch + validate M click(id: ID): string // error or "" fill(id: ID, value: string): string hover(id: ID): string assert(id: ID, condition: Condition, value?: string): string waitFor(id: ID, condition: WaitCondition, timeoutMs?: number): string screenshot(id?: ID): string // base64 PNG }
```

Every method validates `id ∈ keys(M)` before dispatching to the underlying `Locator.forId(cdp, id)` call. An unknown ID is a compile-time-detectable error in typed environments and a runtime error with a precise message otherwise.

4.5 Security Model

The DOM address map defines the **declared capability surface** of the application. The security model has three layers:

Layer 1 — Build-time exclusion: Elements annotated `data-ddas-address="exclude"` are never emitted to M. Internal debug panels, admin controls, and development-only elements can be excluded from the map before deployment.

Layer 2 — Access-controlled map endpoint: M is served at `/.well-known/ddas-map.json` behind `Authorization: Bearer <token>`. An unauthenticated client can observe `data-ddas-id` attributes in the HTML but cannot enumerate M with its metadata (labels, types, file paths, required flags).

Layer 3 — Scoped agent maps: An agent α receives a subset $M_\alpha \subseteq M$. Interactions with IDs outside M_α are rejected at the agent client layer before any CDP call is made. Scope assignment is external to DOMAddress (handled by the application's auth system).

This model satisfies the **principle of least privilege**: agents interact only with what they are explicitly authorized to interact with, and the authorization is declared at build time.

5 Realization Strategies

The DDAS model (Sections 2–4) is deliberately agnostic to how and when IDs are assigned. This section describes three realization strategies — compile-time, build-time, and runtime — along with their trade-offs. The formal properties (determinism, uniqueness, breaking change semantics, security model) hold under all three strategies provided the ID derivation algorithm is faithfully implemented.

5.1 Compile-Time Assignment

A compiler with access to the component tree assigns IDs during compilation, injecting `data-ddas-id` attributes into the emitted HTML/template output. The map M is emitted as a build artifact alongside the application.

Advantages:

- Determinism is trivially satisfied — the compiler processes the same AST every build
- Compositionality enables incremental recomputation (only reprocess changed files)
- ID assignment errors (Priority 6 — unlabelable elements) are surfaced as compile-time errors
- Zero runtime overhead — IDs are static attributes in the HTML

Requirements:

- The compiler must understand the component model (file paths, component boundaries)
- Suitable for compilers with template analysis: Svelte, Angular AOT, or custom toolchains

Complexity: $O(n)$ in element count per component file.

5.2 Build-Time Assignment (Plugin)

A build tool plugin (Vite, Webpack, esbuild, Rollup) processes the compiled output and injects IDs as a post-build transformation. The plugin scans HTML output, applies the derivation algorithm, and emits the map.

Advantages:

- Framework-agnostic — works with React, Vue, Angular, Svelte, or plain HTML
- Requires no change to the compiler itself — operates on standard HTML output
- Can be added to existing projects without changing the build pipeline

Trade-offs:

- The plugin sees rendered HTML, not the component source — deriving the `feature` and `component` segments requires heuristics or configuration mapping from output files back to source paths
- Priority 6 errors are build warnings, not compile-time errors
- May not see the full component tree if SSR and CSR produce different HTML

5.3 Runtime Assignment

A JavaScript library scans the live DOM at page load and assigns IDs dynamically. The map is generated at runtime and served from the application's memory or a dedicated endpoint.

Advantages:

- No build tooling changes — drop-in library
- Works with any framework, any build tool, any server
- Sees the actual rendered DOM, including dynamically generated elements

Trade-offs:

- Determinism depends on consistent DOM rendering order — framework updates, async rendering, and hydration timing can produce different ID assignments across page loads
- Runtime performance cost — DOM traversal on every page load
- Uniqueness is harder to guarantee in the presence of lazy-loaded components and dynamic rendering
- Priority 6 errors surface at runtime, not at build time
- The map is not a static artifact — it must be generated per session or cached carefully

5.4 Strategy Comparison

No single strategy dominates. Compile-time provides the strongest guarantees but requires compiler integration. Runtime provides the broadest compatibility but weakens determinism. Build-time plugins offer a practical middle ground for existing projects.

5.5 Map Distribution

Regardless of realization strategy, the map M is distributed via a well-known endpoint:

```
GET /.well-known/ddas-map.json
Authorization: Bearer <token>
```

Access requires a bearer token (Section 4.5, Layer 2). In development, the map is regenerated on source change. In production, the map is a static artifact deployed alongside application assets. Agents load M once when they start and validate it against the running application version.

5.6 Agent Interaction Layer

The typed agent API (Section 4.4) consumes M through any existing browser automation framework. On each interaction call, the client:

1. Verifies id $\in$ keys(M)
2. Generates a CSS attribute selector: `[data-ddas-id='<id>']`
3. Delegates to the host automation framework's interaction primitives (click, fill, hover, wait)

No new browser driver or DOM interaction logic is required. The contribution is the contract validation and stable selector generation — not the interaction mechanics. Any automation framework that supports CSS attribute selectors (Playwright, Selenium, Puppeteer, Cypress, or a raw CDP client) can serve as the interaction backend.

---

6 Related Work

6.1 Test Locator Fragility

Leotta et al. (2013) in *"Comparing the Maintainability of Selenium WebDriver Test Suites employing different Locator Strategies"* provide the seminal empirical study of selector fragility. They compare six locator strategies across industrial applications and find that CSS selectors break at a rate of 30–60% per major refactor, while ID-based selectors break at 10–15%. DDAS makes ID-based selectors the default and makes those IDs system-guaranteed.

Leotta et al. (2018) in *"Repairing Browser-Based Tests Using the DOM"* propose automatic repair of broken locators using DOM similarity. This work acknowledges that locator stability is unachievable through repair alone and argues for better authoring practices — which DDAS provides at the tooling level.

Stocco et al. (2018) in *"Visual Web Test Repair"* propose repairing broken tests using visual similarity (screenshot comparison). While effective for visual regressions, this approach cannot distinguish semantic changes from visual changes. DOMAddress makes this distinction explicit through the element type metadata in M.

6.2 Semantic Locators

Playwright (Fokeev et al., 2020) introduces `getByRole`, `getByLabel`, and `getByTestId` as semantic alternatives to CSS selectors. `getByTestId` is the closest prior work to DOMAddress: it targets `data-testid` attributes. However, `data-testid` attributes are manually applied (no compiler enforcement), not hierarchically structured (no page/component context), not versioned (no breaking change detection), and not access-controlled (no capability surface).

Testing Library (Dodds, 2019) promotes `getByRole` and `getByLabelText` as the preferred locator strategy, arguing that testing by ARIA role mirrors how users experience the application. That holds for human-readable tests but is insufficient for AI agents, which need enumerable, stable, uniquely-addressable IDs rather than human-readable queries.

6.3 AI Web Agents

WebArena (Zhou et al., 2023): Benchmark of 812 web tasks across 5 realistic applications. Agents use a combination of DOM structure and screenshots to identify elements. Success rate: 14.9% for the best evaluated agent. The paper identifies element grounding (reliably locating the correct element) as the primary failure mode.

Mind2Web (Deng et al., 2023): 2,000 annotated web tasks across 137 websites. Proposes MindAct, a two-stage model that first ranks candidate elements then predicts actions. Task success: 41.6% (seen sites), 23.5% (unseen). The drop on unseen sites is directly attributable to the lack of a reusable element contract — the model must re-learn element locations for each new site.

SeeAct (Zheng et al., 2024): Uses GPT-4V to predict actions from screenshots, then grounds predictions to DOM elements via a choice-based re-ranking. Achieves 51.1% task success. The paper explicitly identifies the grounding step as the performance bottleneck and calls for *"a stable, machine-readable element addressing layer"* as future infrastructure.

AgentBench (Liu et al., 2023): Multi-domain benchmark including web shopping, OS interaction, and database tasks. Web tasks have the lowest success rate (10.3% average) of all domains. The paper attributes this to the dynamic, unstructured nature of web element addressing.

DOMAddress is the infrastructure primitive that all four works identify as missing. It does not replace agent intelligence — it provides the stable foundation on which intelligent agents can be reliably built.

Table 1: Realization strategy comparison.

Property	Compile-time	Build-time (plugin)	Runtime
Determinism	Guaranteed	Guaranteed (if input HTML is stable)	Conditional (depends on render order)
Compositionality	Yes	Partial (output-level, not source-level)	No
Error detection	Compile-time	Build-time	Runtime
Framework coupling	Requires compiler access	Framework-agnostic	Framework-agnostic
Dynamic elements	Not addressed (static only)	Not addressed (static only)	Naturally handles dynamic elements
Performance overhead	Zero (static attributes)	Zero (static attributes)	DOM traversal per page load
Map as build artifact	Yes	Yes	No (generated per session)

6.4 API Contract Systems

OpenAPI (OpenAPI Initiative, 2015): Machine-readable REST API specification. Enables typed client generation, contract testing, and breaking change detection. The DOM address map is the direct analogue for UI surfaces.

GraphQL introspection (Facebook, 2015): Allows clients to query an API's schema at runtime. The `jsc dom-address export --format openapi` command provides analogous static introspection for UI surfaces.

Consumer-Driven Contract Testing (Fowler, 2011): Testing pattern where API consumers define expected contracts that providers must satisfy. DOMAddress enables consumer-driven contracts for UI surfaces — an agent can declare which IDs it depends on, and the CI gate (`jsc dom-address diff`) verifies that the provider (the application) continues to satisfy the contract.

6.5 WebMCP — Declared Capabilities for Cooperative Websites

WebMCP (Brosset, 2025; Perera, 2025) is a W3C Community Group draft specification that allows websites to expose structured, callable tools to AI agents via the `navigator.modelContext` browser API. Websites register named functions with JSON Schema inputs and typed outputs; agents call these functions directly without interacting with the DOM. Perera (2025) reports a 67.6% reduction in processing requirements and 97.9% task success rate across 1,890 API calls.

WebMCP and DDAS attack the same core problem — making web interaction reliable for AI agents — but from opposite directions:

WebMCP solves the cooperative case: the website developer explicitly declares what agents can do, with full control over the API surface — powerful for first-party agent integrations (e.g., a shopping site exposing `add_to_cart` as a tool), but requiring active developer participation.

DDAS solves the universal case: the build tooling assigns stable identifiers to the existing interactive surface without requiring the developer to declare agent-specific capabilities. This suits applications where the interactive elements *are* the capabilities — forms, buttons, navigation — and the goal is to make them reliably addressable rather than to abstract them behind function calls.

The two approaches are complementary. An agent encountering a WebMCP-enabled site should prefer its declared tools. An agent encountering a DDAS-enabled site should use its element contract. An agent encountering neither falls back to DOM scraping or visual grounding. A complete agent architecture may support all three strategies with graceful degradation.

6.6 Compile-Time UI Analysis

Svelte (Harris, 2019): Compiles component templates at build time, demonstrating that full component tree analysis is practical. The Svelte compiler's knowledge of component structure is exactly what DDAS's ID derivation algorithm requires.

Angular's strict template checking (Google, 2020): Typechecks component templates at compile time, including property bindings and event listeners. This work shows that compiler analysis of HTML templates — including tracking which elements have which attributes — is industrially feasible.

Storybook Autodocs (Storybookjs, n.d.): Extracts component APIs from TypeScript types for human documentation. DDAS extends this to machine-readable contracts for programmatic interaction — replacing human-readable prop tables with agent-consumable element maps.

6.7 Accessibility Tree Selectors as a Stable Interface — and Their Limits

The accessibility community has long argued that semantic markup — ARIA labels, roles, and landmarks (W3C, 2023a; Faulkner et al., 2021) — produces more stable interfaces than structural CSS. This insight directly motivates Playwright's `getByRole` and `getByLabel` strategies, and it is empirically supported: Leotta et al. (2013) find that ARIA-based selectors break at roughly half the rate of CSS selectors across major refactors. WCAG-compliant markup (W3C, 2023b), by virtue of being human-meaningful rather than structural, is less likely to change incidentally during UI refactoring.

DOMAddress shares this insight — label-derived element suffixes draw directly from ARIA attributes and visible text — but accessibility tree selectors fail to satisfy the requirements of AI-native interaction on five distinct dimensions:

D1 — Non-uniqueness. ARIA role and label combinations are not guaranteed to be unique within an application. A page with two "Submit" buttons, two navigation sections labeled "Main", or two forms with an "Email" field exposes ambiguous selectors that resolve to different elements in different contexts. `getByRole("button", { name: "Submit" })` is undefined behavior in any application with multiple submit actions. DOMAddress's hierarchical scheme

Table 2: WebMCP and DDAS compared.

Dimension	WebMCP	DDAS
Approach	Website declares callable functions	Build tooling assigns element IDs
DOM dependency	None — bypasses DOM entirely	Yes — IDs reference real elements
Adoption model	Opt-in per website (developer registers tools)	Universal for apps using DDAS-aware tooling
What agent sees	Functions with typed I/O schemas	Elements with typed descriptors
Interaction model	Function calls returning structured data	Element interactions (click, fill, hover)
Coverage	Only cooperative websites	Any application with DDAS-aware build tooling

(`app:feature:component:element`) provides uniqueness by construction — the component context disambiguates elements that share semantic labels.

D2 — Copy-sensitivity. `aria-label` and visible text are human-facing strings subject to copy changes, internationalisation, and A/B testing. An element addressed as `getByLabel("Pay now")` silently breaks when a copywriter changes the label to "Complete payment" — a change with no effect on the element's identity or behavior. DOMAddress IDs are derived from structural source identity (file path + element type + position), not human-readable strings. Label changes do not affect IDs; they appear in the map as advisory `label-changed` events, not breaking changes.

D3 — Non-enumerability. The ARIA accessibility tree is a runtime construct — it exists only when a browser renders the page. There is no build-time artifact that lists all addressable elements in an application, their types, their required status, or their component context. An AI agent cannot discover the application's interactive surface without loading and traversing a live DOM. DOMAddress emits M at build time, enabling agents to load the full surface before any browser interaction occurs.

D4 — Absence of versioning. No existing accessibility tooling provides a diff between two versions of an application's ARIA surface, nor a semantic classification of which changes are breaking from an agent perspective. A removed `aria-label` is indistinguishable from a changed one at the tooling level. DOMAddress's `jsc dom-address diff` provides exact, classified change sets: removed IDs (breaking), type changes (breaking), label changes (advisory), additions (non-breaking).

D5 — Absence of access control. Accessibility trees are universally readable — any agent with DOM access can enumerate every ARIA-labelled element in an application, including internal admin controls, debug panels, and development scaffolding. DOMAddress's map endpoint is access-controlled; build-time exclusion (`data-ddas-address="exclude"`) removes sensitive elements from the declared surface; and per-agent scoping enforces the principle of least privilege at the element granularity.

These five dimensions — uniqueness, copy-stability, enumerability, versioning, and access control — constitute the gap between accessibility as a human-facing concern and element addressing as a machine-facing contract. DOMAddress closes all five. Accessibility tree selectors, while a meaningful improvement over CSS, address none of them.

It is worth noting that DOMAddress and accessibility are not in competition: DOMAddress uses ARIA metadata as an *input* to ID derivation, and compiler-enforced addressing may encourage better ARIA coverage as a side effect (see Section 10, Future Work). The relationship is synergistic — DOMAddress builds on the accessibility foundation and adds the contract layer that machine consumers require.

---

7 Research Questions

RQ1: How stable are compiler-assigned DOM addresses across behavior-preserving refactors? We hypothesize 100% stability for refactors that do not rename components, move elements between components, or change element semantic roles.

RQ2: What fraction of interactive elements in production applications can be automatically addressed without developer annotation? We hypothesize >85% for elements with labels, names, or ARIA attributes; the remainder require `data-ddas-address` annotation.

RQ3: Does ID validation at the agent client layer eliminate the class of selector-not-found failures observed in current AI agent benchmarks? We hypothesize that all failures of this class become detectable at map-load time rather than interaction time.

RQ4: Does breaking change detection (`jsc dom-address diff`) achieve 100% recall on contract regressions? Recall is 100% by construction for removed IDs and type changes; we investigate false negative rates for label changes that affect agent behavior without changing element type.

RQ5: What is the performance overhead of the ID assignment pass on build times? We expect sub-5% overhead for typical projects given the pass's $O(n)$ complexity.

---

8 Evaluation Plan

8.1 Stability Study (RQ1, RQ2)

Dataset: SignFlow application (31 pages, 240+ interactive elements, 12 months of git history, 47 refactoring commits).

Method: Replay each refactoring commit, run the DOM address pass, compute $|\text{Removed}(M_v, M_{v'})|$ for each consecutive pair. Classify each removal as: (a) behavior-changing (element removed or renamed intentionally), (b) behavior-preserving (element moved within same component), (c) annotation-required (element lacks sufficient metadata for automatic addressing).

Baseline: Compare ID stability against manually maintained `data-testid` attributes in the same codebase.

8.2 Agent Interaction Study (RQ3)

Dataset: 180 agent interactions against SignFlow, covering 6 user journeys (login, document upload, approval flow, payment, settings, logout).

Method: Run each interaction with (a) CSS selectors, (b) Playwright `getByTestId`, (c) DOMAddress `DomAgent`. Measure: selector-not-found failures, wrong-element interactions, total interaction success rate.

Metric: Interaction success rate (interaction completes without element-not-found error).

8.3 Breaking Change Detection Study (RQ4)

Dataset: 47 refactoring commits from the SignFlow history, manually classified as breaking or non-breaking from an agent perspective.

Method: Run `jsc dom-address diff` on each consecutive pair. Compute precision and recall against the manual classification.

8.4 Performance Study (RQ5)

Dataset: Justscript compiler build times for SignFlow across 10 runs before and after the DOM address pass.

Metric: Incremental build time overhead (percentage increase in compile time for a single changed file).

---

9 Limitations and Threats to Validity

Evaluation not yet executed: This draft specifies the evaluation protocol (Section 8) and the hypotheses it tests (Section 7) but does not report results — the SignFlow stability study, the 180-interaction agent study, the breaking-change recall study, and the build-overhead measurement have not yet been run. The companion implementation document tracks this directly: its stability table records expected values with commit counts and recall marked "TBD" pending execution. The formal properties in Section 4 (Theorems 1–2, breaking-change semantics) hold independent of this empirical work, since they follow from the specification rather than from measurement — but the paper's claims about real-world stability, false-positive rates, and recall are hypotheses until the protocol in Section 8 is executed and reported. The companion implementation document does report some preliminary, smaller-scale numbers outside the SignFlow study (e.g., a rough build-overhead figure and interaction success rate from early testing) — these are implementation notes, not results of the protocol specified in Section 8, and should not be read as satisfying RQ1–RQ5 until independently verified at the scale and rigor that protocol specifies.

Dynamic elements: Elements rendered conditionally or generated from data (infinite scroll, data tables) produce IDs that depend on runtime state. DOMAddress addresses this through a `data-ddas-address-key` convention for data-driven elements, but the evaluation is limited to statically-rendered components.

Single application: The primary evaluation is against SignFlow. Generalizability to other application architectures (React SPA, server-rendered HTML, Web Components) is a threat to external validity.

Annotation burden: Elements without labels, names, or ARIA attributes require manual annotation. The annotation rate in production applications is unknown; RQ2 provides a first measurement.

Realization strategy: The formal model is agnostic to whether IDs are assigned at compile time, build time, or runtime (Section 5). Each strategy has different guarantees for determinism and dynamic element support. The evaluation in this paper does not yet compare strategies empirically.

---

10 Future Work

Realization strategy comparison: Empirically compare compile-time, build-time (plugin), and runtime assignment strategies (Section 5) across multiple frameworks (React, Vue, Svelte, Angular) and application architectures (SPA, SSR, hybrid). Measure determinism guarantees, coverage rates, and developer experience for each strategy.

Reference implementations: Develop open-source DDAS implementations for multiple toolchains — compiler plugins (Svelte, Angular AOT), build plugins (Vite, Webpack, Rollup), and a runtime JavaScript library — to validate framework-agnostic portability and to lower the adoption barrier for researchers and practitioners.

Temporal IDs for dynamic elements: Extend the scheme to handle data-driven elements: `app:table:row:{record-id}:edit-button` where `{record-id}` is a stable data identifier rather than a DOM position. Investigate the interaction between runtime data keys and static ID determinism.

LLM grounding integration: Evaluate whether providing M as context to an LLM agent (vs. a DOM dump or screenshot) improves task success rates on WebArena/Mind2Web benchmarks. The hypothesis is that a typed map reduces the grounding search space and eliminates ambiguity.

Agent interaction protocol: Design a typed, map-validated interaction protocol for AI agents consuming DDAS contracts. Such a protocol would sit between agent reasoning and browser automation, providing contract enforcement (ID validation, type compatibility checking, audit logging) as a distinct architectural layer.

Bidirectional contracts: Allow agents to declare the subset of M they depend on, and verify that the application satisfies the declared contract — consumer-driven contract testing for UI surfaces.

Accessibility alignment: Investigate the relationship between DDAS IDs and WCAG compliance: whether systematic element addressing encourages accessible markup as a side effect, given that elements must have labels or names to be automatically addressable.

---

11 Conclusion

Web applications are increasingly consumed by AI agents and automated systems, yet they expose no stable, typed, versioned, access-controlled contract for their interactive surface. CSS selectors and XPath are structurally fragile; ARIA-based semantic selectors are manually maintained and not enumerable; `data-testid` conventions are inconsistently applied and unversioned.

The Deterministic DOM Addressing System addresses this gap by defining a formal model for hierarchical element identification, a

versioned map schema, breaking change semantics, and a capability-based security model. The system is agnostic to realization strategy: IDs may be assigned at compile time, build time, or runtime, with different trade-offs in determinism, coverage, and compatibility (Section 5). The resulting DOM address map is an artifact with the same status as type definitions, API schemas, or database DDL: a machine-readable, versioned, access-controlled contract for the application's interactive surface.

We believe DDAS represents a necessary infrastructure primitive for the emerging AI-native application stack. The pattern — systematic ID assignment, versioned map emission, breaking change detection, and capability-scoped agent access — is independent of any particular compiler, framework, or language. We invite the community to develop reference implementations across toolchains and to investigate the interaction protocol by which agents consume these contracts.

12 References

References are ordered alphabetically by first author surname or organization name, matching the author-year citation style used throughout the text.

Brosset, P. (2025). *AI agents and the web — A proposal to keep developers in the loop.* https://patrickbrosset.com/articles/2025-08-28-ai-agents-and-the-web-a-proposal-to-keep-developers-in-the-loop/.

Deng, X., Gu, Y., Zheng, B., Chen, S., Stevens, S., Wang, B., Sun, H., Su, Y. (2023). *Mind2Web: Towards a Generalist Agent for the Web.* NeurIPS 2023.

Dodds, K.C. (2019). *Testing Library.* https://testing-library.com.

Facebook. (2015). *GraphQL: A query language for your API.* https://graphql.org.

Faulkner, S., Eicholz, A., Leithead, T., Danilo, A. (2021). *HTML Living Standard — ARIA in HTML.* WHATWG.

Fokeev, A., Lerner, Y., Lipatov, D. (2020). *Playwright: Fast and reliable end-to-end testing for modern web apps.* Microsoft. https://playwright.dev.

Fowler, M. (2011). *Consumer-Driven Contracts: A Service Evolution Pattern.* martinfowler.com.

Google. (2020). *Angular — Template Type Checking.* https://angular.dev/tools/cli/template-typecheck.

Harris, R. (2019). *Svelte: Cybernetically enhanced web apps.* https://svelte.dev.

Huggins, J. (2004). *Selenium: Browser automation.* ThoughtWorks.

Leotta, M., Clerissi, D., Ricca, F., Tonella, P. (2013). *Comparing the Maintainability of Selenium WebDriver Test Suites employing different Locator Strategies.* ICST 2013.

Leotta, M., Stocco, A., Ricca, F., Tonella, P. (2018). *Repairing Browser-Based Tests Using the DOM.* ICST 2018.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., Tang, J. (2023). *AgentBench: Evaluating LLMs as Agents.* arXiv:2308.03688.

Méndez, F. (2017). *AsyncAPI Specification.* Now maintained by the AsyncAPI Initiative (Linux Foundation). https://www.asyncapi.com.

OpenAPI Initiative. (2015). *OpenAPI Specification.* https://spec.openapis.org.

OXC Project. (2023). *OXC: A collection of high-performance tools for JavaScript and TypeScript written in Rust.* https://oxc.rs.

Perera, D. (2025). *webMCP: Efficient AI-Native Client-Side Interaction for Agent-Ready Web Design.* arXiv:2508.09171.

Stocco, A., Leotta, M., Ricca, F., Tonella, P. (2018). *Visual Web Test Repair.* ISSTA 2018.

Storybookjs. (n.d.). *Storybook Autodocs.* https://storybook.js.org/docs/writing-docs/autodocs. Feature introduced in Storybook's 6.x–7.x releases (2021–2023); exact introduction date not independently confirmed.

Tam, T. (2011). *Swagger Specification.* Wordnik. https://swagger.io.

W3C. (2023a). *Accessible Rich Internet Applications (WAI-ARIA) 1.2.* https://www.w3.org/TR/wai-aria-1.2/.

W3C. (2023b). *Web Content Accessibility Guidelines (WCAG) 2.2.* https://www.w3.org/TR/WCAG22/.

W3C Web Machine Learning Community Group. (2025). *WebMCP Specification.* https://webmachinelearning.github.io/webmcp/.

Zheng, B., Gou, B., Kil, J., Sun, H., Su, Y. (2024). *GPT-4V(ision) is a Generalist Web Agent, if Grounded.* arXiv:2401.01614. (SeeAct)

Zhou, S., Xu, F.F., Zhu, H., Zhou, X., Lo, R., Sridhar, S., Cheng, X., Ou, Y., Bisk, Y., Fried, D., Alon, U., Neubig, G. (2023). *WebArena: A Realistic Web Environment for Building Autonomous Agents.* arXiv:2307.13854.